CTR Developer's Guide

Graphics: Command Lists

Version 2011/06/20

The content of this document is highly confidential and should be handled accordingly.

CTR-06-0036-001-B
Released: June 20, 2011

Table of Contents

Figures

1 Introduction

Nintendo's CTR platform is powered by a "PICA" graphics chip that was developed by Digital Media Professionals (DMP). This document explores methods by which commands can be sent to the PICA chip for processing, and is intended as a supplement to the "DMPGL20_system_API_specificaton.pdf" file. Readers who are new to CTR may want to read this document before reading the above-mentioned DMP document.

Readers are expected to be familiar with basic terms used to discuss 3d graphics and rendering, and should also be comfortable with the basic terminology used in CTR's nngx library. We will, however, highlight terms that may be unfamiliar to readers who have not worked on embedded systems or on DMP-based hardware before. Terms with ***intense emphasis*** will be mainly explained inline. Terms with *subtle emphasis* will be partially explained inline and also explained later on in the document in more detail. Terms that relate to C program code are written in `Courier New` font.

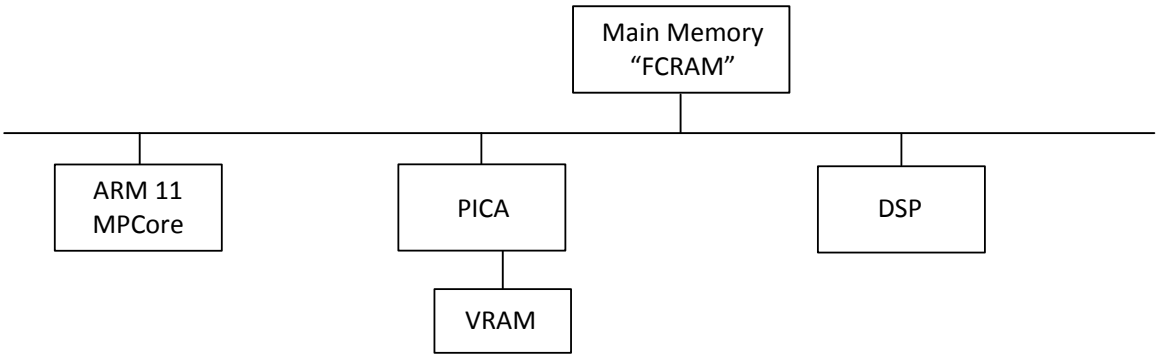
While this document refers specifically to the DMPGL 2.0 rendering API (which is similar to OpenGL), the material herein is also applicable to other Nintendo-provided APIs such as GD, GR, and NintendoWare for CTR.

1.1 Relevant Architectural Details

The PICA graphics chip is connected to the main system bus. It is able to communicate with other clients (such as the ARM11 MPCore CPU and "FCRAM" Main Memory) that sit on the bus. PICA can communicate with other clients in a number of ways. First, portions of PICA's control registers are memory-mapped into the CPU's address space. These registers are only accessible by the graphics driver, and the driver is ultimately in charge of accessing those registers to make requests of PICA. Secondly, PICA can initiate load and store data requests to the bus as needed while generating image data. These requests are all targeted at FCRAM. Finally, PICA can send interrupt signals to the CPU. The signals will be routed to the graphics driver for processing.

Behind the PICA chip is a small area of memory called *VRAM*. All access to VRAM is controlled by the PICA GPU. To move data into or out of VRAM, requests must be made to the PICA graphics chip. Thus PICA acts as the gate-keeper to VRAM and as such is the "owner" of the VRAM hardware.

Figure 1-1 Bus Connections of Various Components in CTR



1.2 Primitives

1.2.1 Command Requests

For CTR, the core unit of execution control at the driver level is called a ***Command Request***. Command Requests are small, fixed-sized chunks of memory that contain high-level driver instructions. These instructions eventually get passed on to the graphics hardware. Upon receiving one of these instructions, the graphics hardware will execute the command.

The types of Command Requests are as follows:

- ***DMA***
  - Instructs the PICA GPU to load a texture or vertex buffer into VRAM from a source outside of VRAM. No conversion is done on the data being loaded, and so the data should be in a "PICA-native" format. PICA will issue the load request to the bus, and manages the complete transfer of the data from FCRAM.
- ***Render***
  - Instruct the PICA GPU to fetch *register update commands* from a source located outside of VRAM. The data contained therein is used to set graphics rendering state in PICA and to kick off drawing of various types of geometry.
- ***Memory Fill***
  - Instructs the PICA GPU to "memset" the color and/or depth buffers in VRAM.
- ***Post-Filter***
  - Instructs the PICA GPU to copy image data stored in VRAM to a location outside of VRAM or to another location in VRAM. During this process PICA will convert the data from the PICA-native format to a non-PICA-native format and [optionally] anti-alias the image, thus rendering it suitable for display on the LCDs or for further conversion in preparation for storage to NAND or SD media. PICA will issue the store request to the bus if necessary, and manages the complete transfer of the data to FCRAM.
- ***Copy Texture***
  - Store PICA-native image data from VRAM to another location in VRAM, or to a location outside of VRAM. No conversion is done on the data being stored. PICA will issue the store request to the bus if necessary, and manages the complete transfer of the data to FCRAM.

As mentioned above, these are the primitives that we send to the GPU. Everything else that PICA needs in order to do rendering is built on top of this foundation.

Notes:

- Each request is approximately six 32-bit words.
- Command Requests are only created by the nngx driver. They may not be generated off-line in a

tool, etc.

1.2.2 Command List Object

When looking at the set of Command Requests, one might come to the conclusion that the majority of OpenGL APIs would have to somehow map to the “Render” command. This is in fact correct. Furthermore, one might guess that the request types other than “Render” probably encompass less than a dozen OpenGL APIs in total. That is indeed the case.

From this, it makes sense that in order to draw a scene one will need a lot of “Render” commands sent to PICA, along with some “DMA” and “Copy Tex” commands, and only a few “Memory Fill” or “Post-Filter” requests.

To handle the sequence of Command Requests that needs to be queued up and sent to PICA, the ***Command List Object*** was created. It contains house-keeping information that the graphics driver needs for basic operation, and it also contains a pointer to an array of fixed-sized Command Request structures. Command List Objects are exposed via the “`nngx*Cmdlist*`” APIs. Programmers are allowed to control the number of Command Request structures within a Command List Object via the “`requestcount`” parameter of `nngxCmdlistStorage`. There is one entry / command request per structure.

Along with the `requestcount`, one can also specify `bufsize`. `bufsize` controls the size of the *3d Command Buffer*, which we’ll explain below.

Please note that it’s important to monitor the resources used in each Command List object via `pname` arguments of `NN_GX_CMDLIST_USED_BUFSIZE` and `NN_GX_CMDLIST_USED_REQCOUNT` to the `nngxGetCmdlistParameteri` API. The necessary sizes of each one may vary drastically based on your approach to rendering. We recommend that you periodically monitor these counts during product development and re-balance these aspects of your Command List Objects as appropriate.

1.2.3 3d Command Buffer

There is an interesting thing to know about the “Render” Command Request. That is: it simply takes a pointer to a memory address and a size (in bytes) to fetch. When PICA executes this command it will DMA the contents of that memory into its command processor wherein the data is interpreted as *Register Update Command* packets. These packets are designed to update one or more 32-bit registers within PICA. Nearly all of these registers control aspects related to 3d rendering such as lighting controls, material settings, vertex shader program and constant uploads, etc. Through use of these packets, one typically updates the registers that control graphics state then kicks off rendering by sending packets that contain vertex information of some sort. Therefore Register Update Command packets are instrumental in achieving work related to 3d rendering.

When the Command List Object was designed, it was decided that it should also contain a pointer to a chunk of RAM where Register Update Command packets could be stored. Since this chunk of RAM is a buffer that will hold these command packets, it is named the 3d Command Buffer. The buffer is very important because it used as the storage area where results of [the vast majority of] OpenGL API calls + their parameters are held after being translated into Register Update Commands for PICA.

There is one 3d Command Buffer per Command List Object.

So as you can see, the 3d Command Buffer is used in conjunction with the “Render” Command Request; each “Render” command points to a location within the 3d Command Buffer where Register Update Command packets have been queued. For a given Command List Object, said buffer is typically the one associated with said object. This is by convention and standard procedure only; exceptions and alternate approaches will be explained later in this document.

1.2.4 Register Update Commands

As mentioned above, these are chunks of data partitioned into packets which load values into PICA’s internal registers. The registers and their payloads are thoroughly described in the “DMPGL20_system_API_specification.pdf” document included in the SDK. The process of loading data into a register is one-way: write-only. No dynamic read-modify-write operation can occur when updating PICAs internal registers. However, a partial-write may take place to a register if one does not wish to update the entire 32-bits of it.

Developers are strongly encouraged to pre-build their own Register Update Command packets and 3d Command Buffers via tools or our APIs as part of the game’s development. By building this data outside the game, less work will be required at runtime since the data will already be in a PICA-native format and ready for rendering.

2 Standard Command List Object Operation

2.1 Accumulate

On CTR, the basic operation of Command List Object management is as follows. OpenGL APIs get translated into either Register Update Command packets (more common) or Command Requests (less common), and placed into the appropriate area of the currently bound Command List Object. That area is either the 3d Command Buffer or the Command Request list, respectively. So long as an OpenGL API translates into a Register Update Command, the driver can simply append it to the 3d Command Buffer. When an OpenGL API occurs that cannot be translated to a Register Update Command packet, the driver does a couple of things. First, it appends a special Register Update Command packet into the 3d Command Buffer which will tell the hardware to generate an interrupt. This is done so that the driver knows when the Render command has completed, thus allowing it to dispatch another Command Request. This is known as a “split” of the 3d Command Buffer. Next, the driver computes the area within its 3d Command Buffer that will be sent as part of the “Render” Command for the queued command packets, and then creates the Render Command Request entry for them. This request is appended onto the Command Request list within the Command List Object. Finally, the appropriate Command Request for the given OpenGL API is appended to the Command List Object’s Command Request list.

When another OpenGL API is encountered that will result in a Register Update Command packet, the packet is simply written to the next available address within the 3d Command Buffer (i.e., the address after the split). The process described above is repeated until either the Command List Object runs out of resources (i.e., the 3d Command Buffer is full or there are no more slots in the Command Request list), or the developer requests that the Command List Object be “run”.

2.2 Run

When a command list is set to “Run”, the driver sequentially walks, over time, through the list of Command Requests and sends each one to PICA. On the initial “Run” request, the driver pops the first queued Command Request packet out of the currently bound Command List Object and sends it to PICA. Once PICA has completed processing that request it sends an interrupt to the driver. Part of the nngx driver will be awoken because of this interrupt, and in the interrupt handler the driver will send (“kick”) the next command in the list to the hardware.

This process continues until either all queued items in the command list have been processed or until the developer requests that the driver “Stop” the Command List Object. It should be noted that the Command List Object tracks which commands in its Command Request list have been run, so that if a “Run” is executed after a “Stop”, playback of the Command Requests resumes where the driver left off.

If a Command List Object is in the “Run” state but has stopped because all Command Requests were already sent to PICA (i.e., this list is exhausted), then the driver will send the next Command Request that comes in as soon as it can. Therefore if an OpenGL API occurs that generates a non-Render

Command Request, then that request will be sent to PICA immediately for processing. Any OpenGL API that results in a Register Update Command packet will simply append the packet(s) into the 3d Command Buffer and no “Render” command will be sent at that time. Only if a “split” occurs (as described earlier) will a “Render” command be sent. This is why immediate mode Command List Object management is not so efficient.

To summarize: “running” a Command List Object is not a blocking operation – it is an asynchronous operation. It is a series of small “send next Command Request to PICA hardware” operations that occur over time.

Figure 2-1 CPU pushes the next Command to the GPU

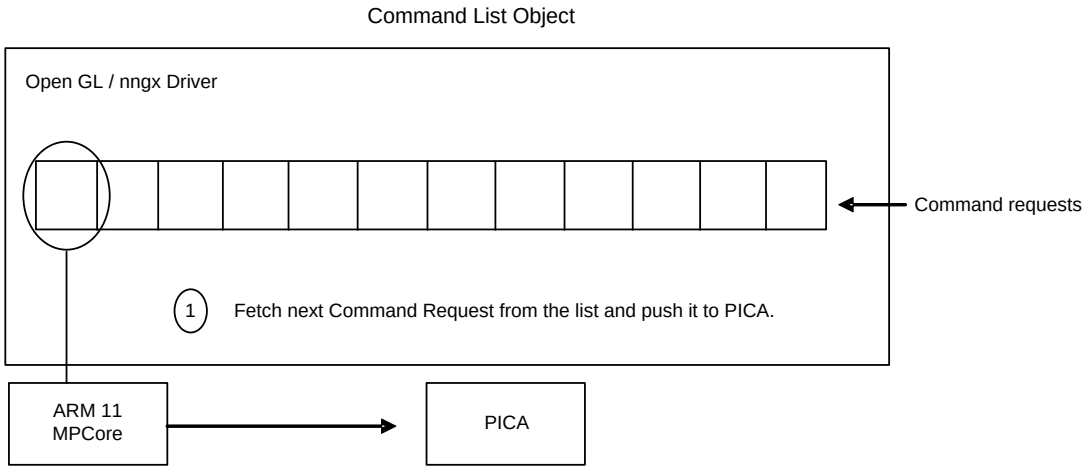


Figure 2-2 GPU processes a non-render Command

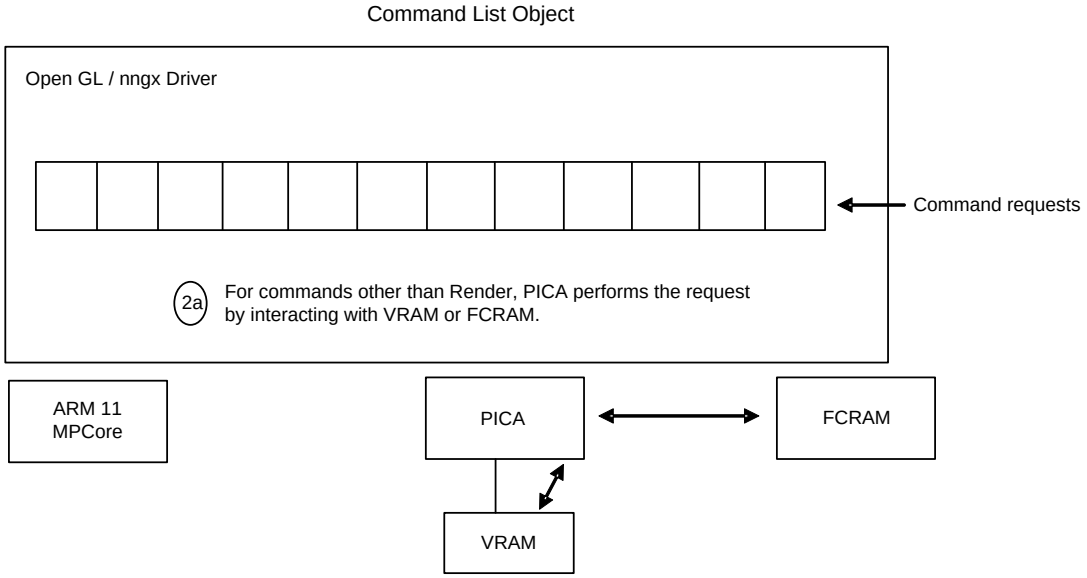


Figure 2-3 GPU processes a Render Command

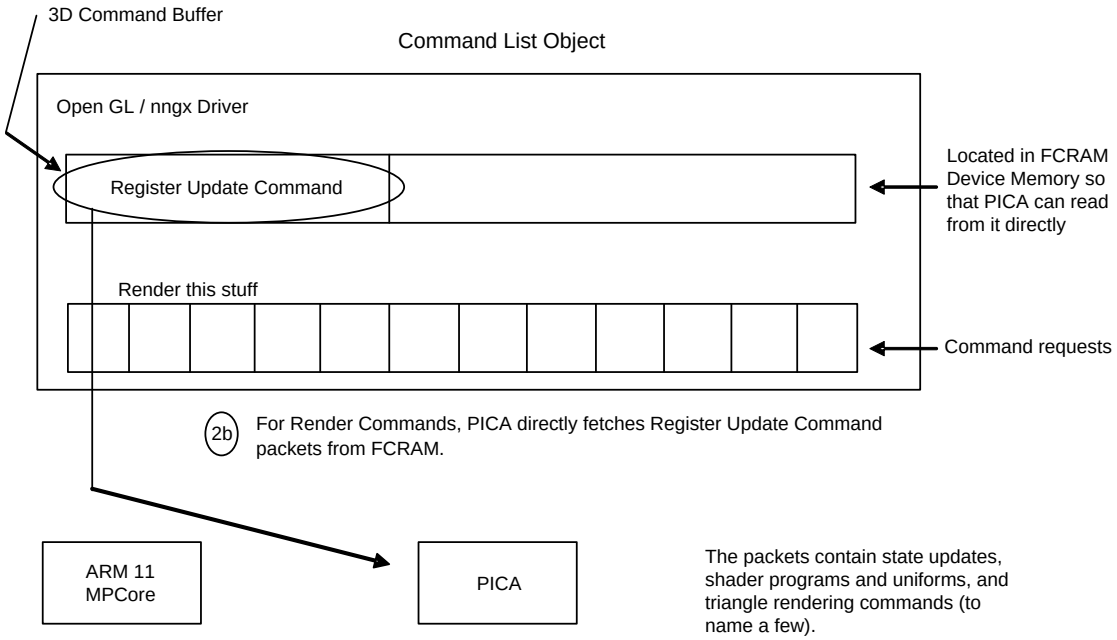


Figure 2-4 GPU is done processing the Command

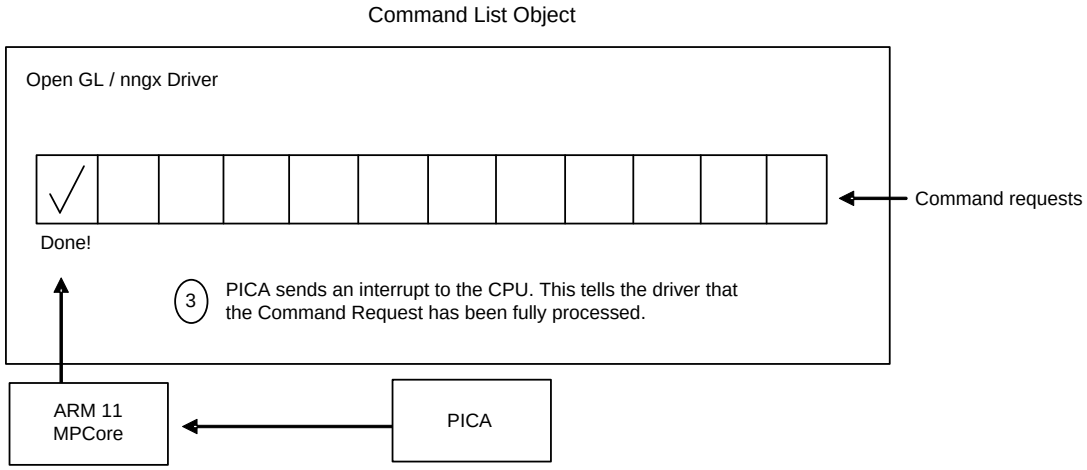
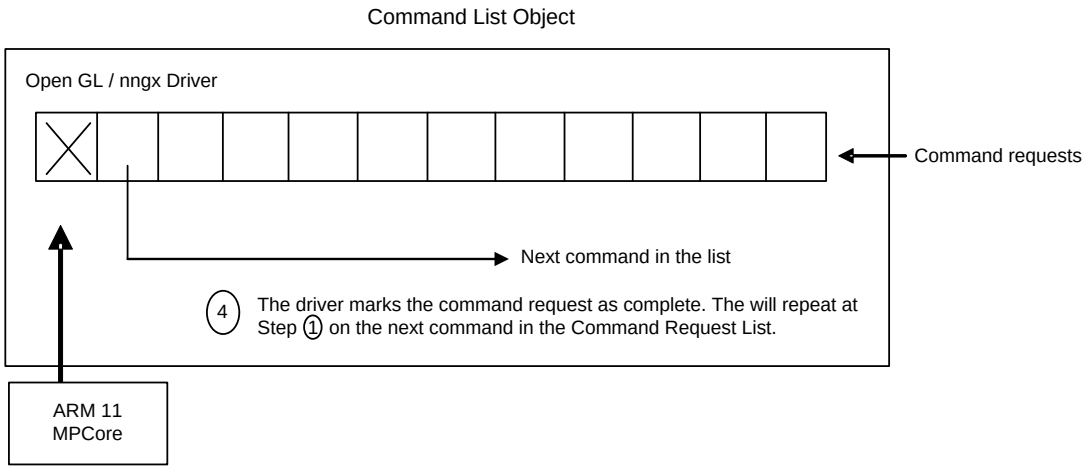


Figure 2-5 The driver prepares to dispatch the next Command



2.3 Clear

This resets all book-keeping data in a Command List Object, thus allowing it to be used as if it was new. One would typically do this once per frame to any Command List Object that is used to manage the majority of rendering. Command List Objects may not be cleared while in the “Run” state.

2.4 Basic list management

The CTR platform provides unique challenges to developers because one has to effectively manage the creation of 3 separate screens worth of data (upper screen left-eye, upper screen right-eye, and lower screen) per frame when ULCD is being supported. When ULCD is not being supported then 2 unique screens worth of data (upper screen, lower screen) must be managed per frame.

There are a couple of options for creating and managing Command List Objects for these screens in your game or application.

Option 1: Immediate mode. A single Command List Object is running while simultaneously accumulating Command Requests. This approach is not terribly good for CPU/GPU concurrency because the GPU may be stalled waiting while the CPU is doing the operations needed to create the next set of Command Requests and Register Update Command packets. Use of well-placed Split commands can help the CPU/GPU concurrency issue, but these are not an ideal solution. Additionally it is not well suited to support dual view rendering for *ULCD* and is generally a poor choice for any title looking to make the best possible use of CPU and GPU time. On the positive side, this option is easy to start out with and doesn’t require as much memory as option #2.

Option 2: Deferred mode. This option requires two (or more) Command List Objects to be managed. During even frames the CPU uses the first Command List Object to create the upcoming frame while the GPU renders commands that were previously accumulated in the second Command List Object. On odd frames, the GPU renders the data that accumulated in the First Command List Object while

the CPU writes the upcoming frame to the second Command List Object. This option is very good for CPU/GPU concurrency because each client can focus on its job and not wait for the other. Note that you still have the issue of what to do for ULCD.

One may wish to dedicate a unique Command List Object to each target screen, or one can try to use only one Command List Object to handle everything. At this time there is no clear answer on what kind of setup to use, but we're leaning towards recommending unique Command List Objects per screen, each double-buffered.

Please note that you must wait until the currently bound Command List Object has completed execution before it will it be safe to switch to a different Command List Object or to copy the color buffer to a display buffer. One can make use of the Command List Object callback system to transition from the processing of one Command List Object to another.

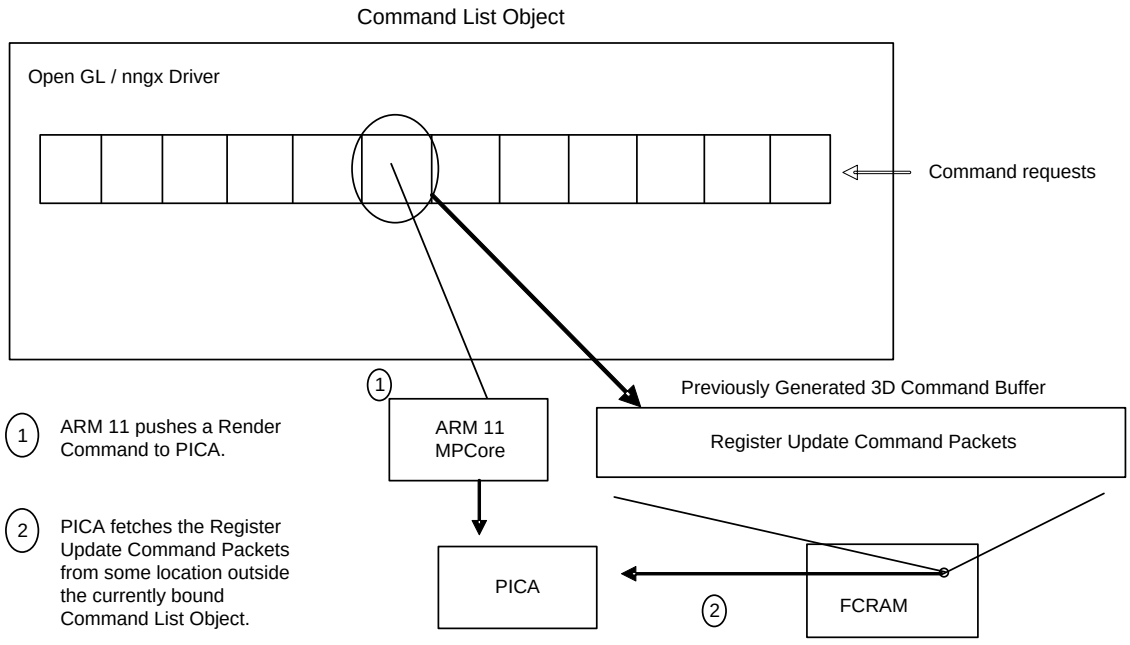
3 Advanced topics

3.1 “Display Lists”

Game developers are generally used to the concept of a “display list”: a chunk of memory that has been loaded with hardware-specific commands for rendering an object. Typically one issues a “branch”-like command to the GPU (or corresponding DMA controller) which will instruct it to detour from the current instruction stream and instead go forth and process data from the display list. Upon completion of processing a display list, the hardware is expected to resume processing at the point just past the branch (much like what happens with function calls on a CPU). Display Lists are often the fastest, most efficient way to draw geometry, and they are easily generated outside the game during the game's development. Developers like to use them because they require minimal processing at runtime.

Since PICA works differently than other rendering hardware you may be familiar with, you have a few options to consider when thinking about the use of Display Lists for CTR. One option is to simply use `nngxAdd3dCommand` with a false `copycmd` parameter. This will insert a “Render” Command Request into the currently bound Command List Object which points to a Register Update Command packet stream of your choosing. However, please be aware that this may not be sufficient to draw complex objects. Complex objects often need material changes or other processing which may involve multiple DMA or a Copy Texture commands. Note that the data being referenced at this address should not be modified until you are certain that PICA has finished consuming it. Also, there is a fixed amount of overhead each time a Render command request is processed by the graphics driver, so be aware that references to external lists should be fairly large in size. This way the overhead is minimized.

Figure 3-1 nngxAdd3dCommand results



In cases where more complex processing is necessary in order to draw an object, one might wish to use a revised approach wherein you manually add all non-render commands necessary for drawing the model into the currently bound Command List Object, and then use one or more `nngxAdd3dCommand` at appropriate places in order to properly draw the model or scene. This method requires a lot of knowledge about the "proper" way to draw the model at runtime and requires more CPU intervention than what one would normally associate with a display list.

The second approach for Display Lists on PICA is to use an entirely separate Command List Object (or equivalently saved data) which contains the entire set of Command Requests + Register Update Command packets in order to draw the model. The nice thing about this method is that it's basically self-contained; it can have everything needed in order to completely draw a model. When one wishes to draw the contents of this kind of display list on PICA, one can wait for the current Command List Object to finish drawing, switch to a different one which contains the relevant data, and then run it. Or, more commonly, one will copy the fixed-sized Command Requests from the display list into their currently-bound Command List Object. This approach of copying Command Requests into the currently bound Command List Object works best when in non-immediate mode, and is not CPU or memory-intensive.

We discuss the copying of data in more detail later in this document.

3.2 GPU State Management

When creating data for a game console or handheld device, one should always attempt to generate as little as possible. The following equation is generally representative: More data = more bandwidth needed + more processing cycles needed = less overall performance. Therefore, in regards to GPU render states it is not advisable to build a full/complete state update into each display list. Conversely, having too little state information in your display lists could very well cause you to end up with corrupted graphics or (worse still) a crashed GPU. So, one should attempt to set only the minimum state necessary in order to draw an object.

Of course, the question then becomes: "What state(s) should I set"? When working directly with the OpenGL Library, this is not so difficult because you can set all states that you need and OpenGL will only update the hardware states that are needed. Unfortunately, we have discovered that there is a significant amount of overhead involved when one uses software to validate state before generating commands for the GPU. This can negatively affect the performance of game titles.

When working with display lists that contain state changes things become a little more difficult because a given display list may not fully configure the hardware, which would result in incorrect rendering. But if one uses display lists only and tries to avoid software state checking of the GPU state, how can one ensure that the hardware is in the correct state before a display list is run?

3.2.1 "Zero State" and deltas

One useful tactic to use when trying to determine what states one should set is to employ the concept of a ***Zero State***. A Zero State is the full set of all GPU render state settings, configured such that each object in the game has very little state delta away from this base state. In other words, the Zero State is the set of states that most objects within your game have in common.

When building display lists for a title that makes use of a Zero State, one will assume that the GPU is already in the Zero State before the display list is called. So all you have to do is export the set of states needed to move from the Zero State to the state needed for your object. This can be considered the ***initial object state***. If the Zero State is well designed, then this set of data is generally minimal.

Additionally, once the object has been drawn but before the Zero State has been restored, the current GPU settings can be considered the ***final object state***. If a second instance is to be drawn immediately after the first, it would be a waste to restore the Zero State, only to re-update the GPU to the object initial state. So it is also prudent to create a ***final to initial object state*** reset state. Finally, when one is done drawing via display list they should restore the hardware to the Zero State so that the next display list can rely on the same assumption that this one did – namely, that the hardware is in the Zero State.

Therefore, when creating and using display lists one will must have both the actual display list and the following sets of the state updates which can be called either before or after the actual display list is called:

- Zero State to initial object state.
- Final object state to Zero State.
- Final object state to initial object state.

Zero-state use can provide significant performance improvements, although there are some downsides. One issue is that a fairly sophisticated toolset is required to generate the data. During game production, changes to assets are quite common and so the optimal Zero State for a given level of the game will most likely change over time, requiring the team to re-generate the ideal Zero State for the area as well as re-update all model data for that area so that they conform to the new Zero State.

3.2.2 SDK Support for state management

CTR SDK has support for generating “full” or “partial” state updates at runtime, in support of saving (and eventual replaying of) of Command List Objects. These, however, refer to only to state potentially modified by the APIs that are called before rendering takes place. They do not refer to the entirety of the GPU render state! Also, please note that these APIs are not [directly] associated with a Zero State concept as mentioned above. However these APIs are useful when attempting to manage runtime creation and playback of Command List Objects, and can be used to help manage the creation of Zero State deltas.

3.2.2.1 Caution 1

One has to be extremely careful when trying to save and play back Command List Objects at runtime since state management can be tricky. Please be aware that each saved Command List Object or saved 3d Command Buffer should not be played back unless the current GPU state is set exactly as it was when the original data was created. Failure to do so could cause errors to occur during rendering or (even worse) cause PICA to crash.

3.2.2.2 Caution 2

When a drawing command takes place, the driver scans its cached state settings and compares each one to see if it is out of sync with what [it thinks] the hardware is set to. If necessary, Register Update Command packets are generated to push all appropriate new or updated settings to the hardware. As of SDK 2.3.4, the DMPGL software routines that do this are larger than the size of the onboard L1 instruction cache. It is therefore considered “expensive” to draw things due to the following factors:

1. The amount of CPU time involved in validating state and pushing appropriate updates to PICA.
2. The amount of CPU time involved in fetching instructions for the validation routines.
3. The fact that the validation routines will effectively wipe out anything your game had in the CPU’s instruction cache. (Hence there will be a time penalty as your code is restored back into the CPU's instruction cache after the drawing and validation code completes.)

This is why we recommend making as few draw calls as possible and making use of state caching. Better yet is to use Zero State and Display Lists extensively during runtime so that this software routine is executed as little as possible.

3.3 Copying data vs. referencing existing data

The decision to copy data for re-submission to PICA as opposed to referencing existing data comes down to a number of factors. Aspects to consider are:

- Parallelism between the CPU and GPU.
- How far ahead in the frame one wishes to do their processing.
- Whether or not the data is being re-submitted to draw a new instance – if so it may need to be modified instead of reused as is.
- The amount of memory available for spare buffers.
- The amount of CPU time and/or bus bandwidth that’s available for data copying.
- Management of data for ULCD rendering.
- Decision for which aspect(s) of a Command List Object should be copied vs. referenced: The 3d Command Buffer or just the Command Requests.
- The number of Command Request slots one has allocated to their currently bound Command List Object.

Let us expand on the list above by noting the following facts:

- Parallelism between the CPU and GPU is most efficient when each can work autonomously. Only by working autonomously can the full potential of the machine be realized. If there is immediate reliance of one upon the other, parallelism will effectively be lost.
- Waiting for a Command List Object to complete execution, then switching execution to a new Command List Object is going to cause the CPU to waste valuable cycles. The CPU will be stalled until the GPU completes all requests. This breaks parallelism.
- Copying of Command Request entries is always done via CPU.
    - Command Request entries are small and fast to copy.
- Copying of 3d Command Buffer data is always done via CPU.

- Copying a 3d Command Buffer and then going back and patching it afterwards is extremely hard on the data cache. To update / patch a Register Update Command packet you must first load it into the data cache, modify it, then write it back out. If the data to be patched is no longer in the cache then it must be reloaded. That takes time and is potentially a waste of bandwidth and CPU + bus cycles.
- Calls into the kernel to flush the CPU's data cache are ***EXTREMELY EXPENSIVE***. One should seek to do this no more than once per frame.

3.4 Other Cautions

When manually creating 3d Command Buffers yourself, you **must** put an Interrupt Generation Command packet at the very end of the packet sequence for each unique “Render” Command Request. Note that this is what the driver does when a “split command” is issued (either explicitly or implicitly from various nngx or OpenGL APIs). Without the interrupt generation, the driver will not know when it is ok to send the next queued Command Request to PICA, and the GPU will appear to have frozen.

Please also note that Register Update Commands fetched by PICA must start on 16-byte aligned address boundaries. Therefore you may need to pad the address of your 3d Command Buffer after issuing an interrupt packet.

Additionally, due to HW errata you must take care when creating Render Command Packets yourself. There are quite a few subtle rules that you must follow in order to prevent malfunction of the PICA chip. Please check the PICA documentation carefully when building your own data. To help the development community, SDSG has created a PC-side library that will emit the proper set of packets for the entirety of PICA. The library, called the Pica Packet Project, can be used as a reference (we include all source code, and it's well documented) or can be integrated into your asset conversion pipelines to assist with creating data for PICA. It is written in .NET managed code (C#). Contact support@noa.com if you would like to evaluate the library

4 Notes about the DMPGL driver

Some developers have stated that the GL driver seems slow, and have asked why this is. The answer is that the GL driver must spend cycles at runtime doing two very important things.

1. Validating graphics state and creating Register Update Command packets as appropriate to program PICA for the desired state.
2. Converting data to PICA-native entities (as appropriate).

Both of these procedures take time. Also, as mentioned above, the code is large and not cache-friendly. If you use the DMPGL APIs, there is little that can be done at runtime to avoid such processing. However, by building Register Update Commands ahead of time (during the production of the game) you can completely avoid the time spent in (2). Use of Zero-state, as well as clever state management can reduce the time needed for (1).

5 Tips for Drawing Stereoscopic Scenes

5.1 History

Drawing the scene for the upper screen in stereo has proven to be challenging. Nintendo's initial guidance was to draw the entire left-eye view, and in the process track where any lighting vector or matrix was sent to the hardware via packets. Next, duplicate the left-eye scene data to create the right-eye scene, and go back and patch the appropriate lighting data & matrices into the right-eye packets.

The above approach has many drawbacks. It proved challenging to ensure that the GPU state at the end of the left-eye view was appropriate when starting to draw the right-eye view, and many teams complained of GPU crashes when starting the draw of the right-eye view. Another downside was that anything that was uploaded to VRAM was effectively uploaded twice (once per eye). Worse, things that *weren't* uploaded to VRAM (being fetched from FCRAM instead) were referenced twice as well. These items generally have low performance characteristics to begin with due to the need to acquire the data from FCRAM, so referencing them twice from FCRAM compounded any performance problems that teams were having. Finally, the act of patching the data really hurts overall performance because non-sequential data needs to be fetched into the CPU's data cache, modified, and then written back to device memory. This is an expensive operation in terms of the amount of time required to perform this task. Afterwards, the CPU data cache much be flushed, and that's even more expensive in terms of time. The net result is that it is very hard to maintain high-performing software when using this approach.

5.2 Improved Method

Our approach to improve performance over the historical method is to try to reduce or eliminate all the downsides of that method. The first major optimization is to remove all patching of data. This means that the left-eye view and the right-eye view must be drawn in an interlaced way so that the CPU will never need to make a second pass over anything that it generates. The benefits of this are substantial, since much time is burned in the previous method due to loading / storing / flushing data through the CPU cache. A bonus side-effect of interlacing the left- and right- eye views is that data will only need to be uploaded into VRAM once since it will be used immediately for both the left- and right- eye drawing. The downside is that one will need to have the color and depth buffers for each eye in VRAM at the same time. Also, rendering algorithms will have to be modified to calculate both the left- and right- eye matrices simultaneously so that the scenes can be drawn in an interlaced manner. (We currently believe that the benefits of this method outweigh the downsides, and in our internal tests the method was quite feasible. It also performed substantially better than the historical method.)

Our tests show that the best performance is reached by:

1. Setting up state for the left-eye view of a triangle mesh, including:

a) Uploading as much texture [and optionally vertex/index] data into VRAM as possible.

b) Setting all appropriate render state, shader programs, etc.

2. Drawing the [sub]mesh (thus creating the left-eye view). This involves issuing packets containing only "draw triangle" type commands, as well as any required post-draw packets. Do not change any GPU state here other than that needed to kick off triangle drawing for one or more meshes.
3. Setting new addresses for the color and depth buffers so that the right-eye render target will be used in the upcoming draw operation (this is a very small set of packets).
4. Update the HW for the right-eye view by sending any VS matrix constants and any other fragment lighting data. The set of packet data here should be fairly small as well.
5. Re-issue the previous draw command(s) from step 2 that were used to kick off rendering (thus now creating the right-eye view).
6. Repeat this process for each [sub]mesh that you draw.

With this approach, no time is spent waiting for the CPU to load/store/flush data from/to FCRAM. This results in a ***significant*** time savings. Additionally, PICA is not being asked to load data into VRAM twice, thus saving cycles both on the GPU side and the CPU side (due to fewer Command Requests being issued in the first place, and fewer state-change packets being fetched into PICA for processing). Some meshes may find a slight additional benefit due to having some of their data already in the vertex or texture caches when the right-eye view is kicked off.

SDSG feels that any teams attempting to improve their rendering performance on CTR should seriously consider such an approach. The approach benefits both the CPU and GPU, and reduces overall load on FCRAM and the main system bus.

6 Summary

In this document we have discussed the internal components to Command List Objects and explored their high-level and low-level processing in an effort to further educate you on the way our platform operates. We have discussed three major techniques (use of Zero State, use of Display Lists, and high-performance ULCD scene management) that are designed to help you get the most out of the available CPU and bus cycles on CTR. With this knowledge you should be armed with a great starting point from which you can begin exploring how to get amazing results out of this particular aspect of Nintendo's CTR platform.

Questions? Comments? Please send us any feedback you have by emailing support@noa.com and using a subject line beginning with: [CDG]

We are happy to take your suggestions for improving our documentation.

Sincerely,
Software Development Support Group
Nintendo of America

Appendix A Glossary

VRAM

Video RAM. That is, random-access memory that is dedicated to storage of data used to generate pictures which change over time. On CTR, VRAM is only accessible via the PICA GPU.

FCRAM

“Fast Cycle RAM”. This refers to a particular manufacturing implementation of random-access memory developed by Fujitsu Semiconductor Limited. It has very good speed of access. On CTR, the main memory consists of FCRAM (as opposed to SRAM, DRAM, or other RAM implementation).

DMA Controller / DMA

Direct Memory Access. DMA refers to the ability of a piece of hardware to access memory on its own, as opposed to requiring an intermediate piece of hardware to perform memory transactions on its behalf.

A DMA Controller is a configurable piece of hardware that can be programmed to load and store memory in specific ways.

DMA (verb)

The act of transferring memory from one location to another without the use of a CPU or other intermediary device.

Command Request

A PICA operation request which is sent to PICA by the nngx driver. These are high-level instructions such as “Render”, “Copy texture out”, “Load Texture in”, “Clear Buffer”, etc.

Register Update Command packet

Entity of 64-bits (or multiples thereof) that contain {address, data} update information for PICA.

3d Command Buffer

A memory buffer owned by a Command List Object. It is used to hold Register Update Command packets.

Command Request list

A memory buffer owned by a Command List Object. It is used to hold Command Requests.

Command List Object

An entity used by the nngx driver to manage processing of the PICA GPU. Games access these via nngx API calls.

Display List

A fixed block of GPU- or system-specific command data. The data is in whatever native format is needed by the hardware.

Display List playback

The act of having a GPU fetch and execute commands stored in a display list. This usually results in an object being drawn.

Zero State

The set of all GPU states that a developer considers to be the “base state” of the GPU for a particular game title. It is used as a reference point against which all other state is set, and if employed should be the state of the GPU both before and after a one or more instances of an object are drawn.

The game would configure the GPU into the Zero State as part of its initialization routine.

Zero State Delta

For any given object, this is the set of state changes needed to get the GPU from the Zero State to the Initial Object State.

Initial Object State

For any given object, this is the [full] state of the GPU immediately before said object is drawn.

Final Object State

For any given object, this is the [full] state of the GPU immediately after said object is drawn.

Final to Initial Object State

For any given object, this is the set of GPU state changes required to reconfigure the GPU from the final object state to the initial object state.

ULCD

Nintendo refers to the upper, stereoscopic LCD display as ULCD when the system is configured to render both left-eye and right-eye views for stereoscopic image display.

Client

A processing device in a system that is connected to the main system bus. Items such as CPUs, GPUs, audio processors, and main memory are all clients that are typically attached to a common bus in a system.

Revision History

Revision Date	Description
2011/06/20	Added information on high-performance rendering techniques.
2010/10/11	Initial version.